Demystifying PROC SQL Join Algorithms

Kirk Paul Lafler, Software Intelligence Corporation, Spring Valley, California

Abstract

When it comes to performing PROC SQL joins, users supply the names of the tables for joining along with the join conditions, and the PROC SQL optimizer determines which of the available join algorithms to use for performing the join operation. Attendees learn about the different types of join algorithms and explore nested loop (brute-force), sort-merge, index, and hash join methods along with selected options to control processing.

Introduction

The SQL procedure is a simple and flexible tool for joining tables of data together. Certainly many of the join techniques can be accomplished using other methods, but the simplicity and flexibility found in the SQL procedure makes it especially interesting, if not indispensable, as a tool for the information practitioner. This paper presents the importance of joins, the join algorithms, how joins are performed without a WHERE clause, with a WHERE clause, using table aliases, and with three tables of data.

Why Join Anyway?

As relational database systems continue to grow in popularity, the need to access normalized data stored in separate tables becomes increasingly important. By relating matching values in key columns in one table with key columns in two or more tables, information can be retrieved as if the data were stored in one huge file. Consequently, the process of joining data from two or more tables can provide new and exciting insights into data relationships.

SQL Joins

A join of two or more tables provides a means of gathering and manipulating data in a single SELECT statement. Joins are specified on a minimum of two tables at a time, where a column from each table is used for the purpose of connecting the two tables using a WHERE clause. Typically, the tables' connecting columns have ***"like"*** values and the same datatype attributes since the join's success is often dependent on these values.

Example Tables

A relational database is simply a collection of tables. Each table contains one or more columns and one or more rows of data. The examples presented in this paper apply an example database consisting of three tables: CUSTOMERS, MOVIES, and ACTORS. Each table appears below.

CUSTOMERS

CUST_NO	NAME	CITY	STATE
11321	John Smith	Miami	FL
44555	Alice Jones	Baltimore	MD
21713	Ryan Adams	Atlanta	GA

MOVIES

CUST_NO	MOVIE_NO	RATING	CATEGORY
44555	1011	PG-13	Adventure
21713	3090	G	Comedy
44555	2198	G	Comedy
37753	4456	PG	Suspense

ACTORS

MOVIE_NO	LEAD_ACTOR
1011	Mel Gibson
2198	Clint Eastwood
3090	Sylvester Stallone

PROC SQL Join Algorithms

When it comes to performing PROC SQL joins, users supply the names of the tables for joining along with the join conditions, and the PROC SQL optimizer determines which of the available join algorithms to use for performing the join operation. There are four algorithms used in the process of performing a join:

- ✓ **Nested Loop** – A nested loop join algorithm may be selected by the SQL optimizer when processing small tables of data where one table is considerably smaller than the other table, the join condition does not contain an equality condition, first row matching is optimized, or using a sort-merge or hash join has been eliminated.

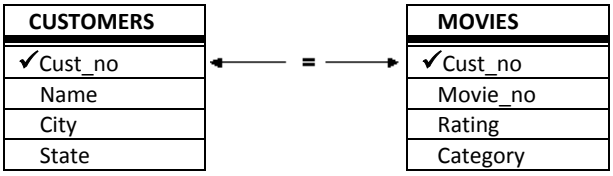
- ✓ **Sort-Merge** – A sort-merge join algorithm may be selected by the SQL optimizer when the tables are small to medium size and an index or hash join algorithm have been eliminated from consideration.

- ✓ **Index** – An index join algorithm may be selected by the SQL optimizer when indexes created on each of the columns participating in the join relationship will improve performance.

- ✓ **Hash** – A hash join algorithm may be selected by the SQL optimizer when sufficient memory is available to the system, and the BUFFERSIZE option is large enough to store the smaller of the tables into memory.

Joining Two Tables With a WHERE Clause

Joining two tables together is a relatively easy process in SQL. To illustrate how a join works, a two-table join is linked in the following diagram.



The following SQL code references a join on two tables with CUST_NO specified as the connecting column.

```
PROC SQL;
 SELECT *
  FROM CUSTOMERS, MOVIES
   WHERE CUSTOMERS.CUST_NO  =
        MOVIES.CUST_NO;
QUIT;
```

In this example, tables CUSTOMERS and MOVIES are used. Each table has a common column, CUST_NO which is used to connect rows together from each when the value of CUST_NO is equal, as specified in the WHERE clause. A WHERE clause restricts what rows of data will be included in the resulting join.

Creating a Cartesian Product

When a WHERE clause is omitted, all possible combinations of rows from each table is produced. This form of join is known as the ***Cartesian Product***. Say for example you join two tables with the first table consisting of 10 rows and the second table with 5 rows. The result of these two tables would consist of 50 rows. Very rarely is there a need to perform a join operation in SQL where a WHERE clause is not specified. The primary importance of being aware of this form of join is to illustrate a base for all joins. Visually, the two tables would be combined without a corresponding WHERE clause as illustrated in the following diagram. Consequently, no connection between common columns exists.

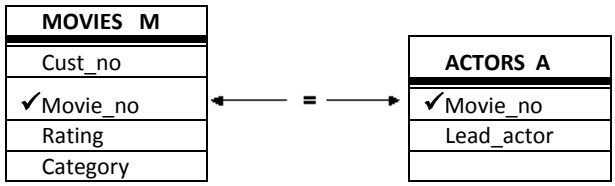


To inspect the results of a Cartesian Product, you could submit the same code as before but without the WHERE clause.

```
PROC SQL;
 SELECT *
   FROM CUSTOMERS, MOVIES;
QUIT;
```

Table Aliases

Table aliases provide a "short-cut" way to reference one or more tables within a join operation. One or more aliases are specified so columns can be selected with a minimal number of keystrokes. To illustrate how table aliases in a join works, a two-table join is linked in the following diagram.



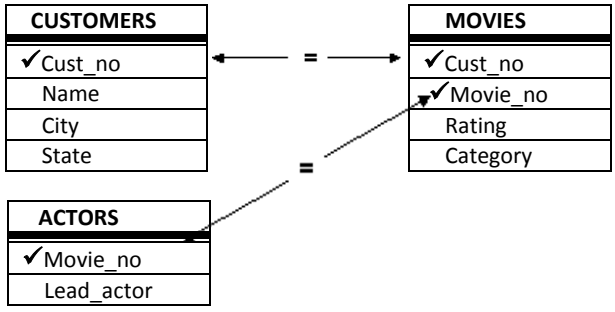
The following SQL code illustrates a join on two tables with MOVIE_NO specified as the connecting column. The table aliases are specified in the SELECT statement as qualified names, the FROM clause, and the WHERE clause.

```
PROC SQL;
 SELECT M.MOVIE_NO,
      M.RATING,
      A.LEADING_ACTOR
   FROM MOVIES M, ACTORS A
    WHERE M.MOVIE_NO =
       A.MOVIE_NO;
QUIT;
```

Joining Three Tables

In an earlier example, you saw where customer information was combined with the movies they rented. You may also want to display the leading actor of each movie along with the other information. To do this, you will need to extract information from three different tables: CUSTOMERS, MOVIES, and ACTORS.

A join with three tables follows the same rules as in a two-table join. Each table will need to be listed in the FROM clause with appropriate restrictions specified in the WHERE clause. To illustrate how a three table join works, the following diagram should be visualized.



The following SQL code references a join on three tables with CUST_NO specified as the connecting column for the CUSTOMERS and MOVIES tables, and MOVIE_NO as the connecting column for the MOVIES and ACTORS tables.

```
PROC SQL;
 SELECT C.CUST_NO,
      M.MOVIE_NO,
      M.RATING,
      M.CATEGORY,
      A.LEADING_ACTOR
  FROM CUSTOMERS C,
      MOVIES M,
      ACTORS A
   WHERE C.CUST_NO = M.CUST_NO AND
       M.MOVIE_NO = A.MOVIE_NO;
QUIT;
```

Outer Joins

Generally a join is a process of relating rows in one table with rows in another. But occasionally, you may want to include rows from one or both tables that have no related rows. This concept is referred to as row preservation and is a significant feature offered by the outer join construct.

There are operational and syntax differences between inner (natural) and outer joins. First, the maximum number of tables that can be specified in an outer join is two (the maximum number of tables that can be specified in an inner join is 32). Like an inner join, an outer join relates rows in both tables. But this is where the similarities end because the result table also includes rows with no related rows from one or both of the tables. This special handling of "matched" and "unmatched" rows of data is what differentiates an outer join from an inner join.

An outer join can accomplish a variety of tasks that would require a great deal of effort using other methods. This is not to say that a process similar to an outer join can not be programmed – it would probably just require more work. Let's take a look at a few tasks that are possible with outer joins:

- List all customer accounts with rentals during the month, including customer accounts with no purchase activity.
- Compute the number of rentals placed by each customer, including customers who have not rented.
- Identify movie renters who rented a movie last month, and those who did not.

Another obvious difference between an outer and inner join is the way the syntax is constructed. Outer joins use keywords such as LEFT JOIN, RIGHT JOIN, and FULL JOIN, and has the WHERE clause replaced with an ON clause. These distinctions help identify outer joins from inner joins.

Finally, specifying a left or right outer join is a matter of choice. Simply put, the only difference between a left and right join is the order of the tables they use to relate rows of data. As such, you can use the two types of outer joins interchangeably and is one based on convenience.

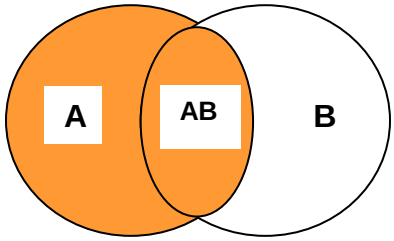
Exploring Outer Joins

Outer joins process data relationships from two tables differently than inner joins. In this section a different type of join, known as an outer join, will be illustrated. The following code example illustrates a left outer join to identify and match movie numbers from the MOVIES and ACTORS tables. The resulting output would contain all rows for which the SQL expression, referenced in the ON clause, matches both tables and retaining all rows from the left table (MOVIES) that did not match any row in the right (ACTORS) table. Essentially the rows from the left table are preserved and captured exactly as they are stored in the table itself, regardless if a match exists.

SQL Code

```
PROC SQL;
 SELECT movies.movie_no, leading_actor, rating
  FROM MOVIES
     LEFT JOIN
      ACTORS
    ON movies.movie_no = actors.movie_no;
QUIT;
```

The result of a Left Outer join is illustrated by the shaded area (A and AB) in the following diagram.

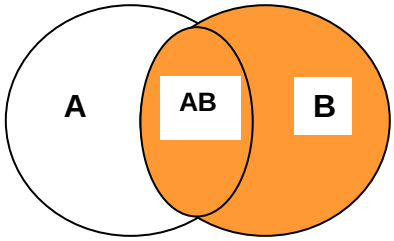


The next example illustrates the result of using a right outer join to identify and match movie titles from the MOVIES and ACTORS tables. The resulting output would contain all rows for which the SQL expression, referenced in the ON clause, matches in both tables (is true) and all rows from the right table (ACTORS) that did not match any row in the left (MOVIES) table.

SQL Code

```
PROC SQL;
 SELECT movies.movie_no, actor_leading, rating
  FROM MOVIES
    RIGHT JOIN
      ACTORS
    ON movies.movie_no = actors.movie_no;
QUIT;
```

The result of a Right Outer join is illustrated by the shaded area (AB and B) in the following diagram.



Conclusion

The SQL procedure provides a powerful way to join two or more tables of data. It's easy to learn and use. More importantly, since the SQL procedure follows the ANSI (American National Standards Institute) guidelines, your knowledge is portable to other platforms and vendor implementations. The simplicity and flexibility of performing joins with the SQL procedure makes it an especially interesting, if not indispensable, tool for the information practitioner.

References

Lafler, Kirk Paul (2012), *"Demystifying PROC SQL Join Algorithms,"* North East SAS® Users Group (NESUG) Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2012), *"Demystifying PROC SQL Join Algorithms,"* MidWest SAS® Users Group (MWSUG) Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2012), *"Demystifying PROC SQL Join Algorithms,"* Western Users of SAS® Software (WUSS) Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2012), *"Powerful, But Sometimes Hard-to-find PROC SQL Features,"* Iowa SAS® Users Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2011), *"Powerful and Sometimes Hard-to-find PROC SQL Features,"* PharmaSUG 2011 Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2010), *"Exploring Powerful Features in PROC SQL,"* SAS Global Forum (SGF) Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2008), *"Kirk's Top Ten Best PROC SQL Tips and Techniques,"* Wisconsin Illinois SAS Users Conference (June 26th, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2008), *"Exploring the Undocumented PROC SQL _METHOD Option,"* Proceedings of the SAS Global Forum (SGF) 2008 Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2007), *"Undocumented and Hard-to-find PROC SQL Features,"* Proceedings of the PharmaSUG 2007 Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul and Ben Cochran (2007), *"A Hands-on Tour Inside the World of PROC SQL Features,"* Proceedings of the SAS Global Forum (SGF) 2007 Conference, Software Intelligence Corporation, Spring Valley, CA, and The Bedford Group, USA.

Lafler, Kirk Paul (2006), *"A Hands-on Tour Inside the World of PROC SQL,"* Proceedings of the 31st Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2005), *"Manipulating Data with PROC SQL,"* Proceedings of the 30th Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.

Lafler, Kirk Paul (2004). *PROC SQL: Beyond the Basics Using SAS*, SAS Institute Inc., Cary, NC, USA.

Lafler, Kirk Paul (2003), "*Undocumented and Hard-to-find PROC SQL Features*," *Proceedings of the Eleventh Annual Western Users of SAS Software Conference*.

Acknowledgments

The author extends a special thanks to Chris Barr and Minh Duong, SCSUG 2012 Conference Chairs, for accepting my abstract and paper, and the SCSUG Executive Committee, SAS Institute, and Conference Leaders for organizing a great conference!

Trademark Citations

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies.

Author Information

Kirk Paul Lafler is consultant and founder of Software Intelligence Corporation and has been using SAS since 1979. He is a SAS Certified Professional, provider of IT consulting services, trainer to SAS users around the world, and sasCommunity.org emeritus Advisory Board member. As the author of five books including PROC SQL: Beyond the Basics Using SAS, 2nd Edition (Spring 2013) and PROC SQL: Beyond the Basics Using SAS (2004), Kirk has written more than five hundred papers and articles, been an Invited speaker and trainer at three hundred-plus SAS International, regional, local, and special-interest user group conferences and meetings, and is the recipient of 22 "Best" contributed paper, hands-on workshop (HOW), and poster awards.

Comments and suggestions can be sent to:

Kirk Paul Lafler
Senior Consultant, Application Developer, Trainer and Author
Software Intelligence Corporation
E-mail: KirkLafler@cs.com
LinkedIn: http://www.linkedin.com/in/KirkPaulLafler
Twitter: @sasNerd